NPS-CS-13-004

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

STATE ESTIMATION OF NON-MONOTONIC, PARTIALLY NON-DETERMINISTIC SOFTWARE WITH SPARSE PROBING USING AN UNSCENTED KALMAN FILTER COMBINED WITH LOGIC REASONING

by

Doron Drusinsky
January 2013

Approved for public release; distribution is unlimited

Prepared for: Office of Naval Research (ONR) -
One Liberty Center, 875 North Randolph Street, Suite 1425
Arlington, VA 22203-1995

THIS PAGE INTENTIONALLY LEFT BLANK

NAVAL POSTGRADUATE SCHOOL
Monterey, California 93943-5000

RDML Jan E. Tighe
Interim President

Douglas A. Hensler
Provost

The report entitled *"State Estimation of Non-monotonic, Partially Non-deterministic Software with Sparse Probing using an Unscented Kalman Filter combined with Logic Reasoning"* was prepared for and funded by the Office of Naval Research (ONR), One Liberty Center, 875 North Randolph Street, Suite 1425, Arlington, VA 22203-1995.

Further distribution of all or part of this report is authorized.

This report was prepared by:

Doron Drusinsky, Associate Professor
Computer Science Department

Reviewed by:

Peter J. Denning, Chairman
Computer Science Department

Released by:

Jeffrey D. Paduan
Dean of Research

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
January 2013	Technical Report	1 July 2012 – 31 December 2013

4. TITLE AND SUBTITLE

State Estimation of Non-monotonic, Partially Non-deterministic Software with Sparse Probing using an Unscented Kalman Filter combined with Logic Reasoning

5a. CONTRACT NUMBER
N0001412AF00002

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Doron Drusinsky

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
1411 Cunningham Road
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

NPS-CS-13-004

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Code 822, Office of Naval Research (ONR) - One Liberty Center, 875 North Randolph Street, Suite 1425 , Arlington, VA

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES
The views expressed in this report are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

14. ABSTRACT

This report describes a technique for assessing the state of a general-purpose system using partial probing. The technique utilizes an Unscented Kalman Filter (UKF) combined with in-process and post-process reasoning.
While Kalman Filters (KF) Extended Kalman Filres (EKF), and UKF are typically applied to state-space systems, where an underlying theory provides the a-priori knowledge, this report suggests the application of UKF to monitor general-purpose software systems that do not have an underlying first-principles theory. The suggested technique uses a reasoning component compute the a-priori evaluation.
An important aspect differentiating state-space systems from general-purpose software is that the latter is often concurrent, with a plurality or concurrently executing threads, processes, or devices. As a result, relative execution time of these components (and the derivative state space) is for all intents and purposes non-deterministic. In addition, the suggested technique enables monitoring with probing that is sparse in time and space namely, probing that occurs only one in n cycles or probing that only probes a subset of the software-systems state-space.

15. SUBJECT TERMS
Kalman Filter, state estimation, probing

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
					Doron Drusinsky
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**	UU	32	**19b. TELEPHONE NUMBER** *(include area code)*
Unclassified	Unclassified	Unclassified			831 656 2168

THIS PAGE INTENTIONALLY LEFT BLANK

1. INTRODUCTION

State estimation is often used in process control and performance monitoring applications, where there are many uncertainties to deal with, such as model uncertainties, measurement uncertainties and various noise sources. In such an environment, representing the model state by an (approximated) probability density function (pdf) enables the propagation the pdf of the system states over time, often in some optimal way. It is most common to use the Gaussian pdf, as characterized by its mean and covariance, to represent the model state, process and measurement noises. It is well known that the Kalman Filter (KF) propagates the mean and covariance of the pdf of the model state of linear dynamic systems in an optimal (minimum mean square error) way [1].

In practice, real-world processes are rarely purely linear. The Extended Kalman Filter (EKF) is a well-known technique for applying the KF to nonlinear system. In the EKF, the pdf is propagated through a linear approximation of the system around the operating point at each time instant, using the Jacobian matrices. Jacobians however, are often difficult to calculate, especially in the case of time-sensitive applications. In addition, the EKF's linear system-approximation often introduces state-estimation errors that diverge over time.

Ulmann et-al developed the Unscented Kalman Filter (UKF) [2], which operates on non-linear systems too. The UKF propagates the pdf in a simple and effective way and it is accurate up to second order in estimating mean and covariance.

This report describes the application of the UKF combined with logic-reasoning to the monitoring of general-purpose software using sparse probing. General-purpose software is typically not the type of system usually being monitored using KF techniques because: (i) it lacks well formed state-equations based on first-principles, (ii) its state space often suffers from abrupt changes and is non-monotonic, and (iii) many general-purpose systems, such as multi-threaded systems, are non-deterministic.

We decided to use the UKF because it incorporates the use of general-purpose a-priori knowledge. We use logic reasoning within the a-priori next state UKF computation.

The rest of the paper is organized as follows. Section 2 provides an overview of KF and EKF techniques. Section 3 provides an overview of the UKF technique. Section 4 describes the problem domain and the suggested UKF-based solution using the UKF.

2. THE KALMAN FILTER AND EXTENDED KALMAN FILTER

All forms of the KF (i.e., KF, EKF, UKF, etc.) operate in two phases per iteration: the prediction phase and correction phase, as depicted in Figure 1. The prediction phase uses prediction formulas, usually based on first-principles (*a-priori* knowledge), to predict the estimated state. An example of a first-principles formula used for this phase is the representation of a moving vehicle's state (its location and velocity) at time *t* as a function of the state at time *t*-1, using the laws of motion. Key to the KF

techniques is that one need not record more than a single (or a limited) history of system states.

In the measurement state, the KF integrates *a-posteriori* knowledge in the form of actual measurements; for the moving vehicle example a measurement is in the form of a GPS state approximation received by the satellite triangulation.

The components of the KF estimation process of Figure 1 are:

- A time-update phase, based on a-priori information given as linear update equations.
- A measurement-update phase, where measurements are incorporated into the compound state-estimate; it is based on the Kalman gain which represents the relative proportion of the measurement and time-update components that are used in the state-estimate output

The beauty of the KF set of techniques is that it integrates both forms of knowledge in an optimal way, resulting in a state estimate that is superior to the estimate generated by the a-priori or a-posteriori information alone.

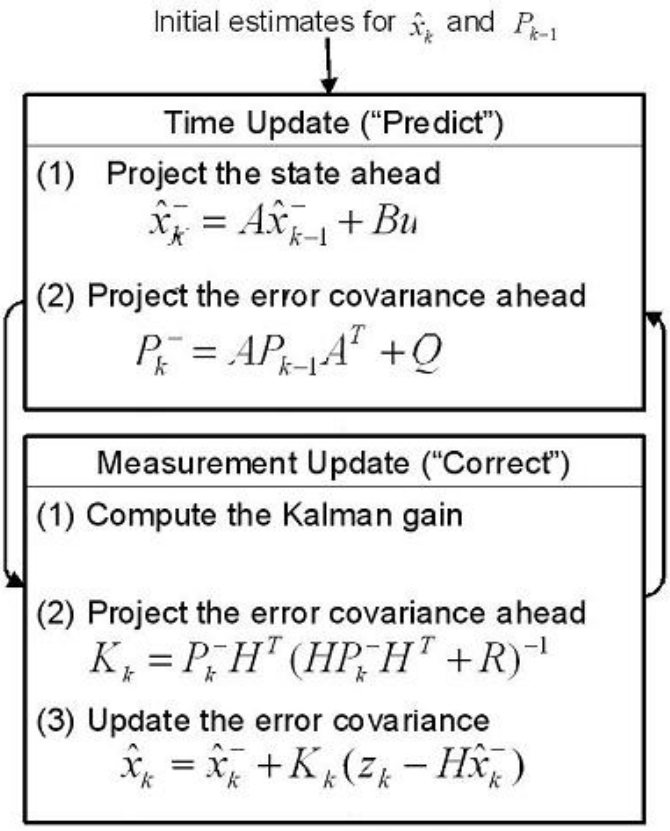


Figure 1. The Kalman Filter

The Kalman Filter is designed for optimal estimation of linear systems. Most interesting real-life system estimation however, are concerned with non-linear systems.

The Extended Kalman Filter (EKF) operates on non linear but differentiable transfer functions. The EKF operates by approximating the state distribution as a Gaussian random variable (GRV) and then propagating it through the first-order linearization of the nonlinear system.

The update equations for EKF are[1]:

Predict

Predicted state estimate	$\hat{\boldsymbol{x}}_{k\mid k-1} = f(\hat{\boldsymbol{x}}_{k-1\mid k-1}, \boldsymbol{u}_{k-1})$
Predicted covariance estimate	$\boldsymbol{P}_{k\mid k-1} = \boldsymbol{F}_{k-1}\boldsymbol{P}_{k-1\mid k-1}\boldsymbol{F}_{k-1}^{\top} + \boldsymbol{Q}_{k-1}$

Update

Innovation or measurement residual	$\tilde{\boldsymbol{y}}_k = \boldsymbol{z}_k - h(\hat{\boldsymbol{x}}_{k\mid k-1})$
Innovation (or residual) covariance	$\boldsymbol{S}_k = \boldsymbol{H}_k\boldsymbol{P}_{k\mid k-1}\boldsymbol{H}_k^{\top} + \boldsymbol{R}_k$
Near-optimal Kalman gain	$\boldsymbol{K}_k = \boldsymbol{P}_{k\mid k-1}\boldsymbol{H}_k^{\top}\boldsymbol{S}_k^{-1}$
Updated state estimate	$\hat{\boldsymbol{x}}_{k\mid k} = \hat{\boldsymbol{x}}_{k\mid k-1} + \boldsymbol{K}_k\tilde{\boldsymbol{y}}_k$
Updated estimate covariance	$\boldsymbol{P}_{k\mid k} = (\boldsymbol{I} - \boldsymbol{K}_k\boldsymbol{H}_k)\boldsymbol{P}_{k\mid k-1}$

where the state transition and observation matrices are defined to be the following Jacobians:

$$\boldsymbol{F}_{k-1} = \left.\frac{\partial f}{\partial \boldsymbol{x}}\right|_{\hat{\boldsymbol{x}}_{k-1\mid k-1}, \boldsymbol{u}_{k-1}}$$

$$\boldsymbol{H}_k = \left.\frac{\partial h}{\partial \boldsymbol{x}}\right|_{\hat{\boldsymbol{x}}_{k\mid k-1}}$$

Unlike the KF, the EKF is not generally optimal when applied to non-linear systems optimal estimator. In addition, if the original estimation is wrong then the estimator often diverges quickly. Also, calculating the Jacobian is computationally expensive.

3. THE UNSCENTED KALMAN FILTER

While EKF approximates state distribution using Gaussian Random Variables (GRV) which are propagated through a first-order linearization of the non-linear system. The UKF [2] replaced this technique with a deterministic sampling approach, where the GRV is represented using a minimal set of chosen sample points. While EKF achieves first order accuracy the UKF captures the posterior mean and covariance accurately to the 2nd order Taylor series expansion, as demonstrated by Jullier and Ullman [2]. The

[1] http://en.wikipedia.org/wiki/Extended_Kalman_filter

UKF's basic framework involves the state estimation of a discrete-time nonlinear dynamic system,

$$x_{k+1} = F(x_k, u_k, v_k)$$

$$y_k = H(x_k, n_k)$$

where:

- x_k represents the unobserved system-state
- u_k is a known exogenous input
- y_k is the observed measurement signal
- v_k is the process noise and n_k is the observation noise.

The block-diagram of Figure 2 depicts such a system. State estimation is concerned with estimating x_k as a Random Variable (RV). It is typically assumed x_k is Gaussian; hence state-estimation is about estimating its mean and covariance.

The Unsecented Transform (UT) is a method for calculating statistics of a RV which undergors a nonlinear transformation [2]. Suppose a RV x of dimension L is propagated through a nonlinear function $y = f(x)$, and that x has mean $\bar{x}$ and covariance **Px**. We create a matrix X of $2L+1$ sigma vectors X_i as follows:

$$X_0 = \bar{x}$$

$$X_i = \bar{x} + (\sqrt{L + \lambda}\,\mathbf{Px})_i \text{ for } i=1,\ldots,L$$

$$X_i = \bar{x} - (\sqrt{L + \lambda}\,\mathbf{Px})_{i-L} \text{ for } i=L+1,\ldots,2L$$

Where $\lambda = \alpha^2 (L+\kappa)-L$ is a scaling parameter, and α determines the spread of the sigma points around $\bar{x}$. κ is also a scaling parameter usually set to 0 or $3-L$.

These sigma points are propagated through the non-linear function $f()$, resulting in $Y_i = f(X_i)$ The mean and covariance of Y are then approximated using a weighted sample mean and covariance of the posterior sigma points,

$$\bar{y} = \sum_{i=0}^{2L} W_i^{(m)} Y_i$$

$$\mathbf{Py} = \sum_{i=0}^{2L} W_i^{(c)} \{Y_i - \bar{y}\}\{Y_i - \bar{y}\}^T$$

Where the weights are given by:

$$W_0^{(m)} = \lambda / (L+\lambda)$$

$$W_0^{(c)} = \lambda / (L+\lambda) + (1-\alpha^2 + \beta)$$

$$Wi^{(m)} = Wi^{(c)} = 1/\{2 (L+\lambda)\} \; i = 1,\ldots 2L.$$

Figure 3 is a block-diagram illustrating the UT process.

The UKF extends UT to the recursive estimation of sigma points but where the state RV is redefined as the concatenation of the original state RV and the noise variables:

$x_k^a = [x_k^T \; v_k^T \; n_k^T]$. Sigma points are selected for the new augmented state RV, instead of for x_k alone.

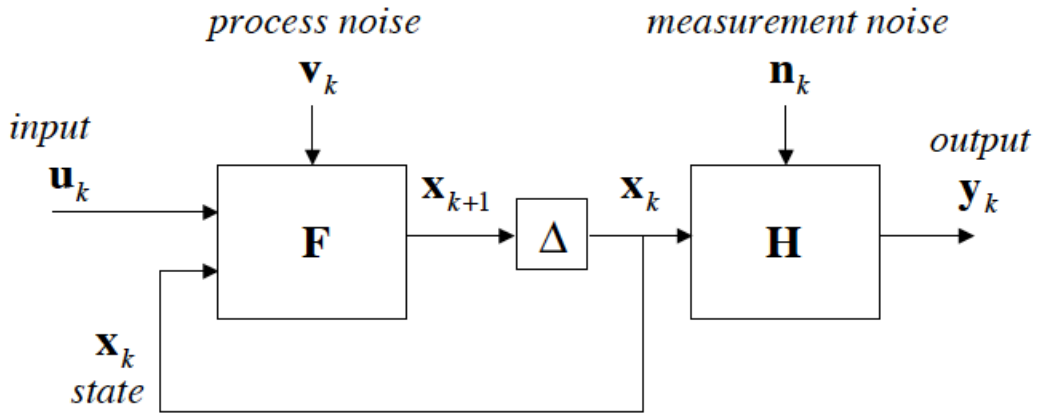


Figure 2. Block diagram of a discrete-time nonlinear dynamic system [2].

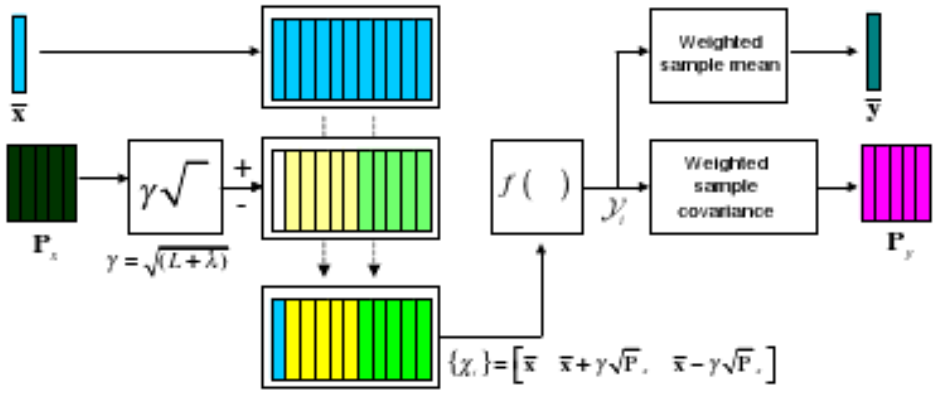


Figure 3. Block diagram of the UT [2]

Java code for the UKF is provided in Appendix B.

4. MONITORING GENERAL PURPOSE SOFTWARE WITH SPARSE PROBING

Applying a KF derivative to *general-purpose* software is untraditional for the following reasons:

- Most often is no first-principles formula that can be used to compute the a-priori knowledge. We use reasoning instead.
- The state space often changes abruptly and non-monotonically.
- Many general-purpose software systems are not fully deterministic, as in the case of multi-threaded software; the game example described in the sequel is such an example.

Nevertheless, we show that the UKF can be used to monitor the state of such systems, using logic reasoning for the UKF's a-priori knowledge propagation component.

In addition to our interest in general-purpose software systems, we are interested in monitoring systems using probing that is sparse in time and space. Temporally sparse probing is one that occurs at a lower frequency than the a-priori knowledge propagation frequency as determined by the logic-reasoning component. Spatially sparse probing probes a subset of the state space. Clearly, when probing general-purpose systems whose state space contains a large volume of state variables, it is not always possible to probe all such states.

4.1. AN EXAMPLE: A TWO PLAYER GAME

The example of general-purpose software used in this report is the following two-player game, called the *Shooting Game*. A player is a computation thread running concurrently to the other with a relative execution cycle drift of +/- *d*%. To gain points, each player takes virtual shots constrained by the following rules.

- A virtual shot taken by player *i* always counts towards that players number of shooting attempts – denoted as *Att(i).* A virtual shot is sometimes also counted as a hit, denoted as *Hit(i).*
- When player *i* makes a shot in counts as *c(n, i)* hits, where *c(n, i)* = *n*/100 * (*abs(Att(j) – Hit(i))*) > 1 ? 1: 0), and *n* is determined by the stage of the game, as discussed below. We call this operation *shoot and increment*, although a player might not actually increment his or her count, depending on *f*. *Shoot and decrement* is defined in a dual manner.
- Initially, both players shoot and increment using *n*=50. The first player accumulate two hits acquires the (unique) lock.
- The player that owns the lock shoots and increments with *n*=75. If s/he misses s/he shoots and decrements *n*=25.

- The player that does *not* own the lock and is waiting for the lock (i.e., s/he has 2 hits) gets a 5% chance to flip the lock, i.e., to own the lock while resetting the opponent's hit count of (get the lock and reset the opponent) - the chance is based on the reading of a millisecond timer.
- A player with 6 or more hits, shoots with n=80-(m%2==1 ? 0: 40) chance shots, where m is time in milliseconds.
- Once player i reaches 10 hits s/he wins the game provided that $3*Hit(j)>Att(i)$. Otherwise, the player starts over the game, i.e., his or hers $Hit(j)$ and $Att(i)$ values are reset to 0.

This game also represents a situation in which the a general-purpose software being monitored is not fully deterministic, with the relative execution time cycle of a player drifts +/- d% relatively to the other player.

Code for the Player game is available in Appendix A.

4.2. UKF REASONING COMPONENT FOR THE SHOOTING GAME

Our UKF-based state estimation system estimates the state of the shooting game by probing, *Hit*(1), *Att*(1), and *Hit*(2). Note that *Hit*(1) depends on *Att*(2) which is not probed. This is an example of spatially sparse probing. We also assume probing is temporally sparse.

We use reasoning within the *f()* component of the UT transform of Figure 3, the component responsible for the a-priori transformation. The reasoning component for the game is written in Java as a collection of Propositional Logic rules. The rules were derived directly from the game specification. Two examples of such a rule are:

- *If Player 2 (named Ted) has less than two hits then increment his count according to c(50, 2).*
- *If Ted has between 6 and 10 hits then increment his count according to c(60, 2).* Note that the game specification for this range is more elaborate, with a non deterministic choice between *c(80, 2)* and *c(40, 2);* since one cannot reason accurately about non-deterministic events we chose a fair coin toss approach, using *c(60, 2).*

Java code for the first reasoning rule is:

```
if (Math.floor(nNumberofHits_TedPS)  < 2) {
       boolean b = shootTed(50, nNumberofHits_EdPS, nNumberOfAttempts_EdPS,
       nNumberofHits_TedPS);
       if (b) {
              nNumberofHits_TedNS++;
        }
}
```

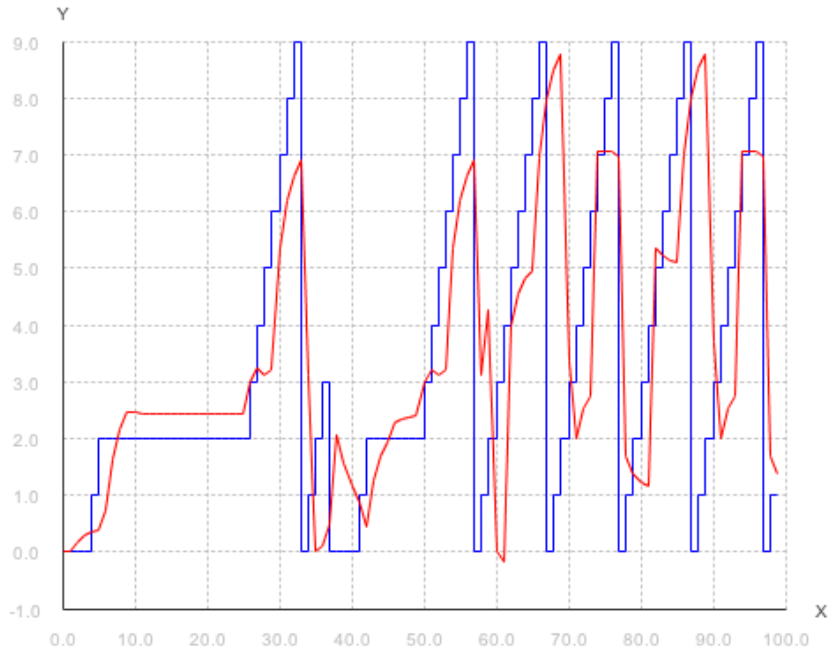
The Java code for the reasoning component is provided in Appendix D. Several important factors distinguish the reasoning component from the implementation of a Player:

- Reasoning is not necessarily complete. For example, our reasoning does not reason about the lock. One of many reasons for doing so was that when both players have two hits then non-deterministic execution time determines which player obtains the lock. Another example is the absence of reasoning about non-deterministic timer based choices within the game, as discussed per the second reasoning rule above.
- Reasoning can be performed on probed state variables (*Hit*(1), *Att*(1), and *Hit*(2)), and un-probed state variables (*Att*(2)) alike. Note that as prescribed by the UKF, the reasoning component generates $2L+1$ (where $L=3$) transformations of the state vector per game cycle, while only a single transform the non-probed variable is required per game cycle.

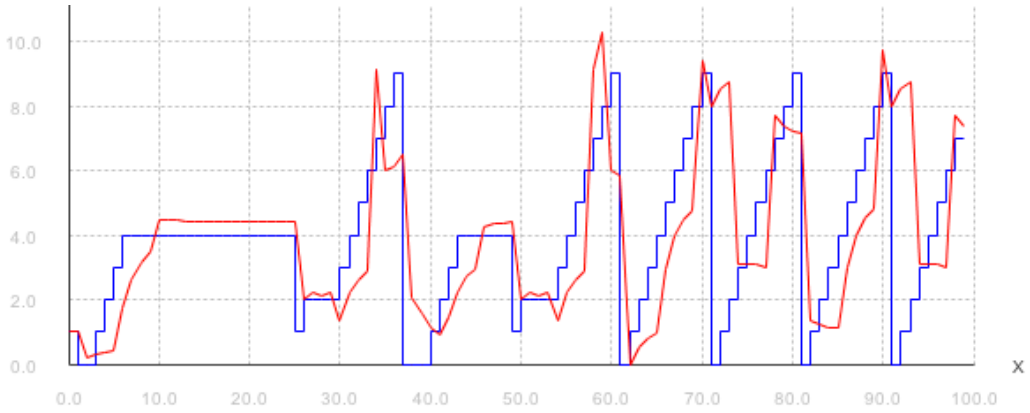
4.3. SHOOTING STATE ESTIMATION RESULTS

Clearly, we cannot analyze the overall Mean Square Error (MSE) of the proposed approach, because it depends heavily on the nature of the software system being monitored (e.g. the degree of non-determinism within the program) and the quality of the reasoning component. Hence we just show somewhat anecdotal results for the shooting game.

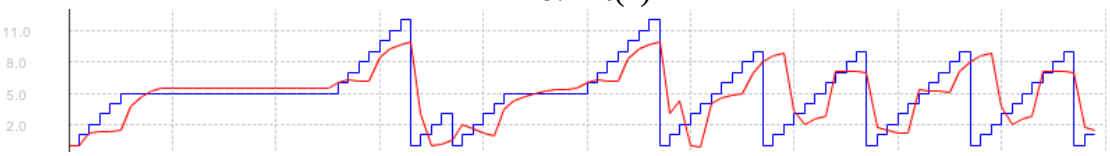
We played the game with 100 cycles per player. We performed temporally sparse probing for all 3 probed values (i.e., *Hit*(1), *Att*(10, and *Hit*(2)), where probing was performed once per K=4 cycles of the game performance. The graphs of Figure 4 demonstrate the effectiveness of the suggested approach.



a. Hit(1)



b. Hit(2)



c. Att(1)

Figure 4. True (blue) vs. UKF (red) state estimates for the Player game

Accuracy results are as follows:

- `RMSE of estimate for Hit(1) is 2.7306244242593904; when normalized it is 0.3049048607132327`
- `RMSE of measurement for Hit(1) is 3.1352830813181765; when normalized it is 0.39191038516477206`

- `RMSE of estimate for Hit(2)  is 2.6694017499675726; when normalized it is 0.26019228200858496`
- `RMSE of measurement for Hit(2) is 3.1064449134018135; when normalized it is 0.3883056141752267`

- `RMSE of estimate for Att(1) is 2.956916917326075; when normalized it is 0.29347430505348177`
- `RMSE of measurement for Att(1) is 3.2526911934581184; when normalized it is 0.4065863991822648`

Note the role of the UKF α variable, which is responsible for adjusting the sigma point distance from the mean.

4.4.POST UKF REASONING VS UKF-REASONING

Post UKF reasoning is reasoning that is not embedded in the UKF loop. As discussed earlier, UKF reasoning, which is performed within the *f*() method of Figure 3, is performed on each of the 2*L*+1 sigma points, i.e., points that are approximately a standard deviation away from the mean. In contrast, post-UKF reasoning is perfumed after a UKF cycle ends. Some examples of post-UKF reasoning for the shooting game are:

- *If the measurement drops to zero from a previous cycle then prefer the reasoning over the UKF computed value.*
- *If UKF computed value is smaller than zero then use zero instead.*

5. CONCLUSION

We have demonstrated an effective approach for state estimation of general-purpose software in which there is no a-priory first principles state equation, the software is possibly non-deterministic, and probing is sparse in space and time. Our approach uses the UKF augmented with propositional logic reasoning with a domain of discourse that contains both the measured (probed) state variables within the UKF and the unmeasured ones.

Given the nature of the sponsored research, we were unable to specific assumptions about the nature of the system under-estimation. We hope to provide more specific technique when we focus on more specific types of software systems.

6. CONTINUING RESEARCH

We plan on developing and demonstrating a technique for monitoring time series properties pertaining-to or asserting-about) the statistics of a software system being estimated via the above mentioned technique. Since the underlying system states are actually random variables, one can only reason about them probabilistically. Our planned technique will allow the specification of properties using existing formal specification techniques and methodologies, yet using a monitoring technique that caters for random variables rather than deterministic propositions.

For example, consider the shooting-game property P1: "*Hit(1) + Hit(2) <= 2 for no more than 3 consecutive cycles*". To monitor this temporal property we can use he following, existing techniques:

1. Represent the property as a formal specification using a specification language such as Statechart assertions, and then monitor the measured values using a corresponding runtime verification tool, such as the StateRover [4, 5].
2. Represent and monitor the property as in (1) but monitor the mean value of the estimated (hidden) values rather than measured values.
3. Represent the property as in (1 and 2) but monitor the mean and covariance values. Monitoring in this case is not deterministic, which will require a new approach, to be developed.

Clearly, approaches (2) and (3) monitor values that are more accurate than the values monitored by approach (1), resulting in more accurate monitoring. Approach (3) is more powerful than (2) in the following sense. Suppose the mean values of Hit(1) and Hit(2) conform to P1 but other values, which are rather probable (e.g. probability of 0.7) do not conform to P1. In such a case, the end user observing the monitored results might be interested to know that such a possibility exists (the possibility and its likelihood).

7. REFERENCES

[1] A. H. Jazwinski, ”Stochastic Processes and Filtering Theory”, Mathematics in Science and Engineering, vol. 64, Academic Press, NewYork and London, 1970.

[2] S. Julier and J. K. Uhlmann, ”Unscented filtering and nonlinear estimation”, Proceedings of the IEEE, vol. 92, 2004, pp.401-422.

[3] E. Wan and R. van der Merwe, The Unscented Kalman Filter, In Kalman Filtering and Neural Networks, Editor(s): Simon Haykin, John Wiley and Sons, 2002, ISBN: 9780471369981

[4] D. Drusinsky, Modeling and Verification Using UML Statecharts, A Working Guide to Reactive System Design, Runtime Monitoring and Execution-based Model Checking. Elsevier, 2006. ISBN: 978-0-7506-7949-7

[5] D. Drusinsky, Practical UML-based Specification, Validation, and Verification of Mission-critical Software. Space Exploration and Defense Software Examples in Practice. Dog Ear Publishing, 2010, ISBN: 978-145750-494-5.

8. APPENDIX A. THE GAME

```
package game;
import java.util.Calendar;
import java.util.Random;
import stateestimation.GameObserver;
/**
 *
 * @author dorondru
*/

public class Player implements Runnable {
       public final static int PLAYERS_INVOCATION_SKEW = 5; // from -2 to 2
       public static final double RATIO = 3;
    Random random;
    Random randomMillisForDebug;
    Random rSleepSkew;
    private boolean bFinished;
              int nNoOfCycles;
              boolean bPauseForDebug;
              public static final int ED = 1;
              public static final int TED = 2;

    private final int nID;
    private Player friend;
    private MyLock sharedLock;
    private int  nNumberOfHits = 0;
    private int  nNumberOfAttempts = 0;
    private int nNumberOfWins = 0;
    private boolean singleCycle2getMillisec = false;

    public Player(int nID, Random rSleepSkew, int nNoOfCycles, MyLock
sharedLock) {
        this.nID = nID;
        random = new Random(100);
        randomMillisForDebug = new Random(150);
        this.rSleepSkew = rSleepSkew;
        this.nNoOfCycles = nNoOfCycles;
        this.sharedLock = sharedLock;
        bFinished = false;
        resetMe(0,"construction", true);
        nNoOfCycles = -1; // temporary
        this.bPauseForDebug = false;
    }

    public void setFriend(Player friend) {
        this.friend = friend;
    }


    public int getID() {
        return this.nID;
    }

    public MyLock getSharedLock() {
       return sharedLock;
    }

    public int getNumberOfHits() {
       return nNumberOfHits;
    }
```

```
    public int getNumberOfAttempts() {
       return nNumberOfAttempts;
    }

    public int getNumberOfWins() {
       return nNumberOfWins;
    }

    public boolean getSingleCycle2getMillisec() {
       return singleCycle2getMillisec;
    }

    @Override
    public void run() {
        for (int i = 0; i < nNoOfCycles; i++) {
           try {
                while (bPauseForDebug) {
                       Thread.sleep(10);
                }
                int nSleepSkew = 0;
                if (rSleepSkew != null) {
                       nSleepSkew = rSleepSkew.nextInt(PLAYERS_INVOCATION_SKEW)
- PLAYERS_INVOCATION_SKEW/2; //[-2,2]
                }
               Thread.sleep(10+ nSleepSkew);
            } catch (InterruptedException e) {}
            singleCycle(i);

        }
        bFinished = true; // for the case I won
    }


    boolean isFinished() {
       return bFinished;
    }


    // implements single cycle in game
    public void singleCycle(int nCycle) {

      try {
       printDebug(nCycle, "---nNumberOfHits=" + nNumberOfHits + ";
nNumberOfAttempts=" + nNumberOfAttempts + "; SingleCycle #" + nCycle + "- nID="
+ nID);
       singleCycle2getMillisec = false;
       if (nNumberOfHits < 2) {
              boolean b = shoot(50, nCycle); // 50% chance
              if (b) {
                     nNumberOfHits++;
                     printDebug2(nCycle,"incrementing #1:  nNumberOfHits=" +
nNumberOfHits + ";- nID=" + nID);
              }
       }
       else if (nNumberOfHits >= 2 && nNumberOfHits < 6) {
              boolean lockIsMine = false;
              if (nNumberOfHits == 2) {
              sharedLock.tryLock(nID);
              printDebug2(nCycle,"tring to get lock - nID=" + nID);
              }
              lockIsMine = sharedLock.isLockOwnedByThisPlayer(nID);
              if (lockIsMine) {
```

```
                    printDebug2(nCycle,"LockIsMine - nID=" + nID);
                           boolean b = shoot(75, nCycle);
                           if (b) {
                                  nNumberOfHits++;
                                  printDebug2(nCycle,"incrementing #2:
nNumberOfHits=" + nNumberOfHits + ";- nID=" + nID);
                           } else {
                                  b = shoot(25, nCycle);
                                  if (b && nNumberOfHits>0) nNumberOfHits--;
                           }
             } else { // I don't have the lock
                    printDebug2(nCycle,"Lock is NOT Mine - nID=" + nID);
                    boolean b = false;
                    if (!b) { //  I missed -  check time in millis
                           long millis = getMillisec(nCycle);
                           if (millis < 0) millis = 0-millis;// unexpected
                           singleCycle2getMillisec = true;
                           int n5 = (int)(millis%100);
                           if (n5 < 5) {
                                  sharedLock.flipLock(nID);
                                  this.friend.resetMe(nCycle, "Reset due to 5%
chance lock flip", false);
                                  printDebug2(nCycle,"Incrementing my friends
count - I'm nID=" + nID);
                           }
                    }
             }
          } // (nNumberOfHits >= 2 && nNumberOfHits < 6)
       else if (nNumberOfHits >= 6 && nNumberOfHits <= 10) {// here
nNumberOfHits = 6 or 0, but 0 will be treated in the next cycle
             printDebug2(nCycle,"nNumberOfHits between 6 and 10=" +
nNumberOfHits + " - nID=" + nID);
             if (nNumberOfHits == 6) {
                    printDebug2(nCycle,"unlocking lock (if locked) - nID=" +
nID);
                    unlock();
             }
             singleCycle2getMillisec = true;
             long millis = getMillisec(nCycle);
             int m = millis%2==1?0:40;
             boolean b = shoot(80-m, nCycle); // 80-m% chance
                    if (b) {
                           printDebug2(nCycle,"incrementing:  nNumberOfHits=" +
nNumberOfHits + ";- nID=" + nID);
                           nNumberOfHits++;
                           if (nNumberOfHits == 10) {
                                  int scaledNoOfHits =
(int)((double)nNumberOfHits * RATIO);
                                  if (nNumberOfAttempts < scaledNoOfHits) { //
start over
                                         System.out.println("reset due to
nNumberOfAttempts > scaledNoOfHits for nID=" + nID);
                                         resetMe(nCycle, "reset due to
nNumberOfAttempts > scaledNoOfHits for nID=" + nID, true);
                                  }
                                  else {
                                         //winner: when not learning mode then
finish, otherwise: just do another round of learning
                                         nNumberOfWins++;
                                         resetToGameStart(nCycle);
                                         printDebug2(nCycle, "nID="+nID+ " has
nNumberOfWins="+nNumberOfWins + " wins");
```

```
                                }
                            }
                        }
      }
      } catch (Exception e) {
         System.err.println("Exception in Player.singleCycle(nCycle), for
nCycle=" + nCycle + ";" + e);
      }
    }


    // f(n, i) = n/100 * (abs(#attempts-of-player-j minus #-hits-of-player-i))
>1? true:false
    boolean shoot(int nProbability, int nCycle) {
       nNumberOfAttempts++;
       int nNoAttemptsOtherPlayer = friend.getNumberOfAttempts();
       int nDiff = Math.abs(nNoAttemptsOtherPlayer - nNumberOfHits);
       double d = (double)nDiff * (double)nProbability / 100.0f;
       if (d > 1.0) return true;
       return false;
    }

    synchronized void unlock() {
       sharedLock.unlock();
     }

    void resetToGameStart(int nCycle) {
       resetMe(nCycle,"reseting due to game start - nID=" + nID, true);
       friend.resetMe(nCycle,"reseting due to game start - nID=" + nID, true);
    }

    void resetMe(int nCycle, String sMsg, boolean bResetNumberOfAttempts) {
       try {
             printDebug(nCycle,"------->reseting due to " + sMsg + " - nID=" +
nID);
                   nNumberOfHits = 0;
                   if (bResetNumberOfAttempts) nNumberOfAttempts = 0;
                   if (sharedLock.isLockOwnedByThisPlayer(nID)) unlock();
       } catch (Exception e) {
             System.err.println("Exception in resetMe; " + e);
       }
    }



    private long getMillisec(int nCycle) {
        Calendar lCDateTime = Calendar.getInstance();
        long l = lCDateTime.getTimeInMillis();
        if (GameObserver.LOCK_STEP) l = randomMillisForDebug.nextInt(100);//
capturing random behavior for deterministic debug
       return l;

    }

    void setGameOver() {
       bFinished = true;
    }


    void setPauseForDebug() {
       bPauseForDebug = true;
    }
```

```
    void resetPauseForDebug() {
       bPauseForDebug = false;
    }

    public static void printDebug(int i, String s) {
          System.out.println(s);
    }
    public static void printDebug2(int i, String s) {
          //printDebug(i, s);
    }
}
```

9. APPENDIX B – UKF IN JAVA[2]

```
package stateestimation;

import static java.lang.Math.*;
public class UnscentedKalmanFilter
{

        //Instance variables
        /** Number of states. */
        private int L;

        /** Number of measurements*/
        private int m;

        /** Tunable. */
        private double alpha;

        /** Tunable. */
        private double ki;
        /** Tunable. */
        private double beta;

        /** Scaling factor. */
        private double lambda;

        /** Scaling factor. */
        private double c;
        /** Weights for means. */
        private Matrix Wm;

        /** Weights for covariance. */
        private Matrix Wc;

        //Extensive generall debug: 0 = no debug, 1 = debug on PC, 2 = debug on
brick
       private static final int DEBUG = 0;
       private final boolean DEBUG_LIGHT = false;
        private final boolean DEBUG2 = false; //ukf.ut()
        private final boolean DEBUG3 = false;  //ukf.sigmas()*/

        /**Constructor
         * @param L  number of states, Doron: N in paper=_Paper
         * @param m  number of measurements
         */
        public UnscentedKalmanFilter(int L, int m)
        {
                //Logger.println("Creating UKF for tracking");
                this.L = L;
                this.m = m;
                alpha=1f; // alpha should change according to the target
software system
                ki=0;  //default
                beta=2f;//pow(alpha, 2) -0.9f;  //lower bound -2; 10 5 10000 -
2* -1 0 1 def:2; default, tunable
                lambda=pow(alpha, 2)*(L+ki)-L;
                c=L+lambda;
```

[2]http://code.google.com/p/cats/source/browse/trunk/simulation/particle_filter/GSim/src/GSim/UnscentedKalmanFilter.java?r=651

```
                Wm = new Matrix(1, (2*L+1), 0.5/c);
                Wm.set(0,0,lambda/c);
                Wc=Wm.copy();
                Wc.set(0,0, Wc.get(0,0) + 1 - pow(alpha, 2) + beta);
                c=sqrt(c); }


        /**
         * UKF, Unscented Kalman Filter, for nonlinear dynamic systems.
         * [x, P] = ukf(f,x,P,h,z,Q,R) returns state estimate x and state
covariance P
         * for nonlinear dynamic system (for simplicity, noises are assumed as
additive):
         *           x_k+1 = f(x_k) + w_k
         *           z_k   = h(x_k) + v_k
         * where w ~ N(0,Q) meaning w is gaussian noise with covariance Q and
         *       v ~ N(0,R) meaning v is gaussian noise with covariance R.
         * @param f  function handle for f(x), nonlinear state equations
         * @param x  "a priori" state estimate
         * @param P  "a priori" estimated state covariance
         * @param h  fanction handle for h(x), measurement equation
         * @param z  current measurement
         * @param Q  process noise covariance
         * @param R  measurement noise covariance
         * @return  "a posteriori" state estimate and P: "a posteriori" state
covariance
         */
        public Matrix[] run_ukf(IFunction f, Matrix[] x_and_P, IFunction h,
Matrix z, Matrix Q, Matrix R) throws Exception //Cholesky can throw exception
        {

                //long time_start_run_ukf = System.currentTimeMillis();

                if (DEBUG != 0)
                {       debug("Entering ukf with the following parameters:");
                        debug("Debug: ukf, x dim= " +
x_and_P[0].getRowDimension() + " x " + x_and_P[0].getColumnDimension() + ", x=
");
                        debug(Matlab.MatrixToString( x_and_P[0]) );
                        debug("Debug: ukf, P dim= " +
x_and_P[1].getRowDimension() + " x " + x_and_P[1].getColumnDimension() + ", P=
");
                        debug(Matlab.MatrixToString(x_and_P[1]));
                        debug("Debug: ukf, z dim= " + z.getRowDimension() + " x
" + z.getColumnDimension() + ", z= ");
                        debug(Matlab.MatrixToString(z));
                        debug("Debug: ukf, Q dim= " + Q.getRowDimension() + " x
" + Q.getColumnDimension() + ", Q= ");
                        debug(Matlab.MatrixToString(Q));
                        debug("Debug: ukf, R dim= " + R.getRowDimension() + " x
" + R.getColumnDimension() + ", R= ");
                        debug(Matlab.MatrixToString(R));
                        debug("Debug: ukf, Wm dim= " + Wm.getRowDimension() + "
x " + Wm.getColumnDimension() + ", Wm= ");
                        debug(Matlab.MatrixToString(Wm));
                        debug("Debug: ukf, Wc dim= " + Wc.getRowDimension() + "
x " + Wc.getColumnDimension() + ", Wc= ");
                        debug(Matlab.MatrixToString(Wc));
                        debug("Starting calculations");
                        if (x_and_P[0].getRowDimension() != L ||
x_and_P[0].getColumnDimension() != 1) debug("WARNING: The dimension of the
state vector (matrix) x is incorrect! Expected dim = " + L +" x 1" );
```

```
                        if (x_and_P[1].getRowDimension() != L ||
x_and_P[1].getColumnDimension() != L) debug("WARNING: The dimension of the
state covariance matrix P is incorrect! Expected dim = " + L +" x " + L);
                        if (Q.getRowDimension() != L || Q.getColumnDimension()
!= L) debug("WARNING: The dimension of the covariance of process matrix Q is
incorrect! Expected dim = " + L +" x " + L);
                        if (z.getRowDimension() != m || z.getColumnDimension()
!= 1) debug("WARNING: The dimension of the measurement vector (matrix) z is
incorrect! Expected dim = " + m +" x 1");
                        if (R.getRowDimension() != m || R.getColumnDimension()
!= m) debug("WARNING: The dimension of the covariance of measurement matrix P
is incorrect! Expected dim = " + m +" x " + m);
                }


                Matrix X = sigmas(x_and_P[0],x_and_P[1],c);  //sigma points
around x, NB: c has been set in the constructor

                if (DEBUG != 0)
                {
                        debug("Debug: ukf, X dim= " + X.getRowDimension() + " x
" + X.getColumnDimension() + ", X (after sigmas() )= ");
                        debug(Matlab.MatrixToString(X));
                }


                Matrix[] ut_f_matrices= ut(f,X,Wm,Wc,L,Q);  //unscented
transformation of process
                Matrix x1 = ut_f_matrices[0];
                Matrix X1 = ut_f_matrices[1];
                Matrix P1 = ut_f_matrices[2];
                Matrix X2 = ut_f_matrices[3];

                if (DEBUG != 0)
                {
                        debug("Debug: ukf, x1 dim= " + x1.getRowDimension() + "
x " + x1.getColumnDimension() + ", x1= ");
                        debug(Matlab.MatrixToString(x1));
                        debug("Debug: ukf, X1 dim= " + X1.getRowDimension() + "
x " + X1.getColumnDimension() + ", X1= ");
                        debug(Matlab.MatrixToString(X1));
                        debug("Debug: ukf, P1 dim= " + P1.getRowDimension() + "
x " + P1.getColumnDimension() + ", P1= ");
                        debug(Matlab.MatrixToString(P1));
                        debug("Debug: ukf, X2 dim= " + X2.getRowDimension() + "
x " + X2.getColumnDimension() + ", X2= ");
                        debug(Matlab.MatrixToString(X2));
                }


                Matrix[] ut_h_matrices =ut(h,X1,Wm,Wc,m, R);  //unscented
transformation of measurments
                Matrix z1 = ut_h_matrices[0];
                Matrix Z1 = ut_h_matrices[1];
                Matrix P2 = ut_h_matrices[2];
                Matrix Z2 = ut_h_matrices[3];
                //long time_after_ut_h = System.currentTimeMillis();
                if (DEBUG != 0)
                {
                        debug("Debug: ukf, z1 dim= " + z1.getRowDimension() + "
x " + z1.getColumnDimension() + ", z1= ");
                        debug(Matlab.MatrixToString(z1));
```

```
                        debug("Debug: ukf, Z1 dim= " + Z1.getRowDimension() + "
x " + Z1.getColumnDimension() + ", Z1= ");
                        debug(Matlab.MatrixToString(Z1));
                        debug("Debug: ukf, P2 dim= " + P2.getRowDimension() + "
x " + P2.getColumnDimension() + ", P2= ");
                        debug(Matlab.MatrixToString(P2));
                        debug("Debug: ukf, Z2 dim= " + Z2.getRowDimension() + "
x " + Z2.getColumnDimension() + ", Z2= ");
                        debug(Matlab.MatrixToString(Z2));
                    }

                    Matrix P12 = ( X2.times(Matlab.diagFromColumn(Wc))
).times(Z2.transpose());  //transformed cross-covariance
                    if (DEBUG != 0)
                    {
                        debug("Debug: ukf, P12 dim= " + P12.getRowDimension() +
" x " + P12.getColumnDimension() + ", P12= ");
                        debug(Matlab.MatrixToString(P12));
                    }
Matrix K = Matrix.identity(P12.getRowDimension(), P12.getColumnDimension());
                    try { // Doron - added
                        K = P2.transpose().solve(P12.transpose()).transpose();
                    } catch (Exception e) {
                        // Doron - added -- just keep diag K
                    }


                    if (DEBUG != 0)
                    {
                        debug("Debug: ukf, K dim= " + K.getRowDimension() + " x
" + K.getColumnDimension() + ", K= ");
                        debug(Matlab.MatrixToString(K));
                    }

                    x_and_P[0] = x1.plus( K.times(z.minus(z1)) );  //state update,
                    x_and_P[1] = P1.minus( K.times(P12.transpose()) ); //covariance
update,
if (DEBUG != 0)
                    {
                        debug("Leaving ukf with the following results:");
                        debug("Debug: ukf, x_updated dim= " +
x_and_P[0].getRowDimension() + " x " + x_and_P[0].getColumnDimension() + ",
x_updated= ");
                        debug(Matlab.MatrixToString(x_and_P[0]));
                        debug("Debug: ukf, P_updated dim= " +
x_and_P[1].getRowDimension() + " x " + x_and_P[1].getColumnDimension() + ",
P_updated= ");
                        debug(Matlab.MatrixToString(x_and_P[1]));
                    }
                return x_and_P;
            }//end of ukf()


            /**
             * Unscented Transformation
             * @param f  nonlinear map
             * @param X  sigma points
             * @param Wm  weights for mean
             * @param Wc  weights for covariance
             * @param n  number of outputs of f
             * @param R  additive covariance
             * @return y: transformed mean, Y: transformed sampling points, P:
```

```
transformed covariance, Y1: transformed deviations
         */
        private Matrix[] ut(IFunction func, Matrix X, Matrix Wm, Matrix Wc, int
n, Matrix R) throws Exception
        {
                int L = X.getColumnDimension();
                Matrix y = Matlab.zeros(n,1);
                Matrix Y = Matlab.zeros(n,L);

int X_row_dim = X.getRowDimension();points
                int Y_row_dim = Y.getRowDimension();
                int y_row_dim = y.getRowDimension();
                Matrix row_in_X;
                for (int k=0; k<L; k
                {
                        //for all columns in X, compute fstate for the given
row vector and put the result in Y
                        row_in_X = X.getMatrix(0, X_row_dim-1, k, k);
                        Y.setMatrix(0, Y_row_dim-1, k, k, func.eval(row_in_X)
);
                        y.setMatrix( 0, y_row_dim-1, 0, 0, ( (Y.getMatrix(0,
Y_row_dim-1, k, k)).times(Wm.get(0, k)) ).plus(y)  );

                }

                Matrix Y1 = Y.minus(  y.times(
Matlab.ones(1,Y.getColumnDimension()) )  );

                Matrix P = Y1.times(Matlab.diagFromColumn(Wc));
                P = P.times(Y1.transpose());
                P.plusEquals(R);

                //create output matrix array
                Matrix[] output = {y,Y,P,Y1};
                return output;
        }//end of ut()

        /**
         *
         * Sigma points around reference point
         * @param x  reference point
         * @param P  covariance
         * @param c  coefficient
         * @return Sigma points // Doron: there s.b. 2L+1 (2N+1, using
paper=_Paper terminology) points
         */
        private Matrix sigmas(Matrix x, Matrix P, double c) throws Exception
//Cholesky can throw exception
        {
             Matrix A = new Matrix( Cholesky.cholesky( P.getArray() ) );.
                A = A.times(c);
                A.transpose();

                int n = x.getRowDimension();


                //Create Y
                Matrix Y = new Matrix(n, n, 1);
                for (int j=0; j<n; j++)  //columns
                {
                        Y.setMatrix(0, n-1, j, j, x);
                }
```

```
                //Create X
                Matrix X = new Matrix(n,(1+n+n));

                X.setMatrix(0, n-1, 0, 0, x);

                Matrix Y_plus_A = Y.plus(A);
                X.setMatrix(0, n-1, 1, n, Y_plus_A);

                Matrix Y_minus_A = Y.minus(A);
                X.setMatrix(0, n-1, n+1, n+n, Y_minus_A

                return X;
        }//end of sigmas()


        /**
         * Prints a filter object
         */
        public String toString()
        {

                String s ="";
                s = " L = " + L + "\n m = " + m + "\n";
                s += " alpha = " + alpha + "\n ki = " + ki + "\n beta = " +
beta +
                "\n lambda = " + lambda + "\n c = " + c + "\n Wm = " +
Matlab.MatrixToString(Wm) + " Wc = " + Matlab.MatrixToString(Wc);

                return s;
        }

    private static void debug(String s)
    {
        if (DEBUG == 0)
                return;
        if(DEBUG == 1){
                System.out.println(s);
        }
        else
                Logger.println(s);
    }

}//end of class
```

10. APPENDIX C – REASONING COMPONENT

The reasoning component for the game is written in Java as a collection of Propositional Logic rules. The rules were derived directly from the game specification.

```
public Matrix eval(Matrix x) {
		Matrix output = new
Matrix(x.getRowDimension(),x.getColumnDimension());
		double nNumberofHits_EdPS = x.get(0, 0);
			double nNumberofHits_EdNS = nNumberofHits_EdPS;

		double nNumberOfAttempts_EdPS = x.get(1, 0);
		double nNumberOfAttempts_EdNS = nNumberOfAttempts_EdPS+1;

		double nNumberofHits_TedPS = x.get(2, 0);
		double nNumberofHits_TedNS = nNumberofHits_TedPS;

		if (Math.floor(nNumberofHits_EdPS) > 2 &&
		nNumberofHits_EdPS < 6 && Math.floor(nNumberofHits_TedPS) > 2 &&
		nNumberofHits_TedPS < 6) {
				if (nNumberofHits_TedPS < nNumberofHits_EdPS) {
					nNumberofHits_TedPS = 2.0;
				} else {
					nNumberofHits_EdPS = 2.0;
				}
		}


		// Ted
		     if (Math.floor(nNumberofHits_TedPS)  < 2) {
				boolean b = shootTed(50, nNumberofHits_EdPS,
nNumberOfAttempts_EdPS, nNumberofHits_TedPS);
				if (b) {
					nNumberofHits_TedNS++;
				}
			}
			else if (nNumberofHits_TedPS < 6.0) {
			   //boolean lockIsMine = false; // not modeling lock
			   if (true /*lockIsMine*/) {
					boolean b = shootTed(75,
nNumberofHits_EdPS, nNumberOfAttempts_EdPS, nNumberofHits_TedPS);
					if (b) {
						nNumberofHits_TedNS++;
					} else {
						b = shootTed(25,
nNumberofHits_EdPS, nNumberOfAttempts_EdPS, nNumberofHits_TedPS);
						if (b && nNumberofHits_TedNS > 0)
nNumberofHits_TedNS--;
					}
				} else { // I don't have the lock
					// not implemented in Estimator
				}
			}
			else if (nNumberofHits_TedPS >= 6 &&
Math.ceil(nNumberofHits_TedPS) < 10) {
					boolean b = shootTed(60, nNumberofHits_EdPS,
```

```
nNumberOfAttempts_EdPS, nNumberofHits_TedPS); // using half way between 80 and
40
                                    if (b) nNumberofHits_TedNS++;
                            }
                            else if (Math.ceil(nNumberofHits_TedPS) >= 10) {
                               if (nNumberofHits_TedPS * Player.RATIO >
nNumberOfAttempts_Ted_estimated) { // NOTE inaccuracy s.b ">="
                                        nNumberofHits_TedNS = 0;
                                        nNumberOfAttempts_Ted_estimated = 0;
                                        nNumberofHits_EdNS = 0;
                                        nNumberOfAttempts_EdNS = 0;
                                } else {
                                        nNumberofHits_TedNS = 0;
                                        nNumberOfAttempts_Ted_estimated = 0;
                               }
                            }


                        //== Ed
                         if (Math.floor(nNumberofHits_EdPS)  < 2) {
                                boolean b = shootEd(50, nNumberofHits_EdPS,
nNumberOfAttempts_EdPS, nNumberofHits_TedPS);
                                if (b) {
                                        nNumberofHits_EdNS++;
                                }
                            }
                            else if (nNumberofHits_EdPS < 6.0) {
                               if (nNumberofHits_TedPS  < 2 || nNumberofHits_TedPS
>= 6) {
                                        boolean b = shootEd(75,
nNumberofHits_EdPS, nNumberOfAttempts_EdPS, nNumberofHits_TedPS);
                                        if (b) {
                                                nNumberofHits_EdNS++;
                                        } else {
                                                b = shootEd(25,
nNumberofHits_EdPS, nNumberOfAttempts_EdPS, nNumberofHits_TedPS);
                                                if (b && nNumberofHits_EdNS > 0)
nNumberofHits_EdNS--;
                                        }
                               } else {/*lockIsMine*/// not implemented in Estimator

                                    }
                            }
                            else if (nNumberofHits_EdPS >= 6 &&
Math.ceil(nNumberofHits_EdPS) < 10) {
                                boolean b = shootEd(60, nNumberofHits_EdPS,
nNumberOfAttempts_EdPS, nNumberofHits_TedPS); // using half way between 80 and
40
                                if (b) nNumberofHits_EdNS++;
                            }
                            else if (Math.ceil(nNumberofHits_EdPS) >= 10) {
                               if (nNumberofHits_EdPS  * Player.RATIO >
nNumberOfAttempts_EdNS) { // NOTE inaccuracy s.b ">="
                                        nNumberofHits_EdNS = 0;
                                        nNumberOfAttempts_EdNS = 0;
                                        nNumberofHits_TedNS = 0;
                                        nNumberOfAttempts_Ted_estimated = 0;
                                } else {
                                        nNumberofHits_EdNS = 0;
                                        nNumberOfAttempts_EdNS = 0;
                               }
                            }
```

```
 output.set(0, 0, nNumberofHits_EdNS);
 output.set(1, 0, nNumberOfAttempts_EdNS);
 output.set(2, 0, nNumberofHits_TedNS);

 return output;
}
```

INITIAL DISTRIBUTION LIST

1. Dr. Sukarno Mertoguno, Office of Naval Research
   Office of Naval Research (ONR) -
   One Liberty Center, 875 North Randolph Street, Suite 1425
   Arlington, VA 22203-1995

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Research Sponsored Programs Office, Code 41
   Naval Postgraduate School
   Monterey, CA 93943

4. Professor Doron Drusinsky
   Naval Postgraduate School
   Monterey, California